11 Semantic Web Services

Semantic Web Services focus on extending traditional Web Services such that their meaning is embedded in the syntactical description. A lot of work, especially in academia, is devoted to this space and the current status and achievements will be highlighted in this chapter. In Section 11.1 the reasons for Semantic Web Services and the main extensions to traditional Web Services are introduced. While some efforts are based on the development of Semantic Web languages, other efforts that are introduced in Section 11.2 use alternative approaches. Section 11.3 discusses in detail the current Semantic Web Service approaches that are based on the technologies developed in the Semantic Web community. Two very "hot topics" in the space are discovery and composition, both of which are discussed in Section 11.4. Section 11.6 provides a summary.

11.1 Semantics of Web Services

Semantic Web Services attempt to increase the usefulness of Web Services by extending them with semantic descriptions. The main areas for extension are interface descriptions incorporating semantic annotations, and the modeling of precise state information of Web Services (especially for long-running interactions).

11.1.1 Why Semantic Web Services?

Web Services, as a technology for application-to-application (A2A) integration over the Web, achieved a big step forward by using XML as its fundamental language. XML has revolutionized the way data is exchanged and represented across the Web. It provides a standard language for describing document types in any arbitrary domain, facilitating the sharing of data across different systems and, most notably, the Web. XML is flexible and extensible, allowing users to create their own tags to match their own specific requirements. As a result XML-based languages have been designed for use in many different fields. In the context of Web Services, WSDL is used to describe both interface and implementation details; SOAP is used to define messages sent to and from services while the datatypes, used in the content of the SOAP messages, are defined using XML Schema. Despite its universality, there remain serious deficiencies in XML as the language for Web Service interactions. XML is a language for defining the structure and

syntax of data. It says nothing about the meaning, or semantics, that is associated with the data. This hinders a potential service client from using a particular Web Service because the need remains for a human to be involved to interpret both the XML descriptions of the Web Services (WSDL) as well as the XML descriptions of the data that the Web Services can exchange (XML Schema). The human has to decide if the service matches his or her needs and whether or not he or she can understand the data that should be sent to and received from the service.

As a consequence of the semantic ambiguity inherent in XML descriptions, a number of further problems arise, particularly when Web Services are to be considered as the basis for automated A2A integration. One problem is the location of Web Services for them to cater to a specific capability required by a potential client. The UDDI specification, amongst others, provides for a registry of service descriptions. However, the descriptions are not formalized and are only useful when interpreted by a human reader.

Automated data transformation is another issue. If the data definitions used by the Web Service do not match those used by the potential client, a transformation is required and this is typically encoded using the eXtensible Stylesheet Language (XSLT). For each pairwise transformation between a client and service two XSLT style sheets are required, one for each direction. Both must be hand-coded as there is no automated means for interpreting the data semantics. This has to be repeated for each client having heterogeneous data definitions.

The WSDL specification provides a means for all the publicly available operations offered by a Web Service to be described. In business processes, services are typically required to offer a complex behavioral pattern. RosettaNet is a B2B standard defining inter-company processes, including structure and semantics for business messages and secure transportation of messages over the Internet. RosettaNet defines various Partner Interface Protocols (PIPs). One example is PIP 3A4 for the exchange of purchase order (PO) request and confirmation messages between trading partners. The PIP defines four messages, PO Request, PO Confirmation and two signal messages to acknowledge the receipt of the request and the confirmation respectively. A Web Service conforming to PIP 3A4 needs to be able to define not only the messages and operations but also the control and data flow between the messages. This is not possible using WSDL alone as the definition of control and data flow between WSDL operations is not possible.

Moving on from the previous point, if one Web Service is defined to support the RosettaNet PIP 3A4 B2B protocol and a potential client supports a different B2B standard such as Electronic Business XML (ebXML), then mediation is required between the behaviors defined in terms of the two B2B protocols used by the client and the server respectively. As the public behavior offered by a Web Service is only informally described, there is very limited possibility for automating the mediation task.

11.1.2 Interface vs. Implementation

An enduring problem in computer science is how to model and design software solutions to problems in terms of the problems themselves rather than in terms of the specific computer machinery required. The notion of abstracting away from the underlying computer machinery is the basis for the evolution from programming at the assembly code level, to languages like C and FORTRAN, and further to declarative languages such as LISP, or object-oriented (OO) languages such as Smalltalk, C++ and Java. With OO languages elements in the domain of the problem are modeled as objects that can cooperate together to solve the problem at hand. Objects are typed, have states, and can send and receive messages to and from each other. Another underlying concept in OO programming is the separation of *what* an object can do from *how* the object achieves this functionality. This is provided for by the separation of interface and implementation. The interface is the public face of an object describing the data and behavior that the object makes available to other objects so that they can invoke its functionality.

Web Services also offer the separation of interface from implementation through the WSDL descriptions. The definition part of a WSDL document defines the datatypes, messages and operations that together define how to access the functionality offered by the service. WSDL also has a section for binding descriptions. This defines the Web location at which the service can be accessed in addition to the communication and message protocols that should be used for message exchange. Although, in this respect, Web Services seem similar to objects with separated interface and implementation, a major difference is that Web Services generally do not maintain state. In other words, rather than building on the OO-influenced remote method invocation (RMI), they build rather more on the earlier remote procedure call (RPC) technology.

As the WSDL definitions are fundamental in allowing potential users of a Web Service to determine how to interact with that service, it is imperative that this information be unambiguously defined. There are two aspects to consider, data and behavior. In WSDL, the recommended technology for the definition of datatypes is XML Schema. The WSDL schema itself provides the means for describing behavior. The drawback for Web Services is that as neither aspect of interface description results in unambiguous meaning, computer systems are blocked from being able to interpret and reason over the description. This, in turn, restricts the opportunities for automated Web Service discovery, composition and invocation, leading to a strong motivation for semantic annotations.

11.1.3 Modeling of State

Long-running business processes are quite common in the real world. Airline reservation systems usually allow holds to be placed on seats for a 48-hour period before they must be confirmed. In supply chain management software, an order for goods may be made from customer to supplier. Fulfilling the order might require

parts to be ordered from a third party. The initial purchase order request remains open until a confirmation message can be sent back from the supplier. It is possible that this may take a number of hours or days. Additionally, the conversation between both partners may require several (possibly asynchronous) messages back and forth. In such cases, it makes no sense for the business partner making the request to block its software systems while waiting for a request to be completed. Instead a token is often shared between the trading partners that can be used to identify the state of the process at the provider's end.

As a result, it seems natural for Web Services that may be involved in long-running interactions to support the notion of state. This is sometimes considered as an attempt to map the distributed computing architecture of a specification like CORBA onto the Web, where a "factory" resource can be used to provide identifiers to new or existing sessions on request. However, there is no specific mechanism defined in WSDL for this purpose. In fact there is a strong argument from the Web community that Web Services, as resources available over the Web, remain stateless following the REST architectural style [299]. However the issue of how to deal with long-running Web Services remains. One approach, from the Grid community, is the Web Services Resource Framework (WSRF) [300], a family of specifications that combine a stateful resource to a Web Service through interfaces defined as part of a WSDL service description. WSRF relies on the WS-Addressing [301] specification as a means for providing the target endpoint for a message sent to a Web Service implementing the WSRF interfaces.

For businesses providing a service that, for example, handles purchase order requests, a hybrid approach involving document-style Web Services and business standard specifications is sometimes adopted. For example, if both partners to a business interaction agree to use the RosettaNet[6] specification for handling purchase orders (POs), they may agree to use a common order identification system. The WSDL definition for the service would then define that an incoming PO document should contain a typed XML element representing the PO identifier. The Web Service interface would remain stateless and simply pass received messages to the back-end systems designed to handle large volumes of concurrent PO requests.

Taking the example of the previous paragraph as typical usage of Web Services in stateful interactions, there is a strong motivation for the formalization of industry B2B specifications through the use of ontologies. The Semantic Web Services machinery could then be employed to automatically tackle interoperability issues between such ontologies at the conceptual level. Mappings would still be required but only to be established between concepts in related ontologies rather than on a one-to-one basis between data instances: the main drawback of using XSLT.

---

6. http://www.rosettanet.org/ (Accessed September 10, 2007).

11.2 Alternatives for Capturing Semantics of Web Services

Using the technology of ontologies to extend traditional Web Services is usually mainly targeted at the definition of the data contained in the messages exchanged by Web Services. One of the important characteristics of Semantic Web Services is their potential to improve on the existing Web Service model when it comes to describing behavioral semantics. Web Services are touted as the basis for a new wave of distributed computing over the Web, taking off from where CORBA [302] and DCOM [303] left off for intranet- and extranet-based distributed systems. Two aspects of behavior requiring a formal operational semantics for Web Services are:

- The external interface the Web Service offers to its potential clients
- The internal definition of the composition of other Web Services a particular service might use to achieve its objectives

Especially in the area of long-running Web Services and composition, a set of alternative technologies and formalisms is used to define precisely the meaning of execution. This is discussed in the following.

11.2.1 Finite State Machines

Finite State Machines (FSMs) are one of the oldest techniques [293][294] in computer science for modeling sequential behavior that depends not only on inputs but also on the state a system is in when an input is received. FSMs consist of five elements: states, state transitions, conditions, input events, and output events. A state provides information about something that has already happened. Transitions indicate a change of state and are described by a condition that, when fulfilled, results in the transition. Actions are activities that are performed at a given moment. For example, different types of activities are when a state is entered (entry), when a state is exited (exit), when a transition occurs (transition) and when an action takes place (action). FSMs can be drawn using statecharts or by state transition tables. Both illustrate the state that an FSM can move to given a current state and set of inputs.

Finite state machines are rule-based and thus are suitable for problem-solving algorithms. They are of two types. Deterministic FSMs are those for which, given a state and set of inputs, the next state can be predicted. Non-deterministic FSMs are those where the next state transition cannot be predicted given an initial state and set of inputs, and an unpredictable external event may affect the FSM. Additionally, there are two models defined for state machines, Moore and Mealy. Moore machines are those where outputs are a function of the state only. Mealy machines are those where outputs are a function of both the state and the inputs. In [295], Hendler recognizes FSMs as a potential useful means to model the process model for Web Services. Additionally [296] and [297] propose FSMs as a useful mecha-

nism to model the internal and external behavior of services as a sequence of transitions between states.

11.2.2 Statechart Diagrams

In his paper [298], Harel notes that state machines are a natural medium for describing the dynamic behavior of complex systems where events may occur at run time, affecting the system's execution. However, he draws attention to the drawbacks of using FSMs for complex systems where the number of states may grow exponentially, resulting in unmanageable complexity and illegibility in the FSM diagrams. To counter these problems, he proposes statecharts as an extension of the notion of FSMs to include the concepts of hierarchy, state clustering, modularity and concurrency.

11.2.3 Petri Nets

Petri nets were invented in 1962 by C. A. Petri [304] as a mathematical model for concurrent, asynchronous, parallel behavior in distributed systems. Graphically, Petri nets are represented as bipartite graphs with place nodes, transition nodes and directed arcs (also called edges) that link them. Bipartite graphs contain a set of vertices that can be divided into two distinct disjoint sets such that no edge can have both endpoints connected to the same set. In the case of Petri nets, no edge can have both ends connected to places or both ends connected to transitions. Places can have one or more tokens. A place that is connected to an ingoing edge of a transition is considered an input for the transition. Similarly, a place that is connected to an outgoing edge of a transition is considered an output for the transition. Transitions may fire as long as sufficient tokens are available at the input places. When this happens, the transition is said to be enabled. Firing results in tokens being removed from input places and added to output places.

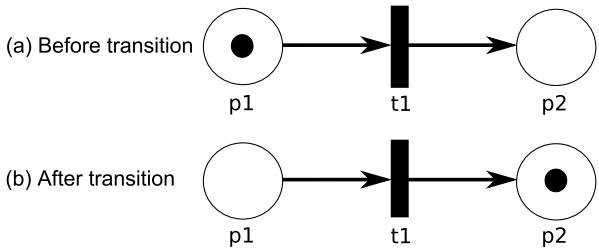

Fig. 11.1. Simple Petri net

Figure 11.1 shows two *markings* (a) and (b) for a simple Petri net with places, p1 and p2, and a transition, t1. A marking defines a possible state of a Petri net by

defining what tokens are available at each of the net's places. In (a), there is a token at place p1 which means the transition, t1, is enabled and may fire. In (b), the transition, t1, has fired and a token has been removed from place p1 and added to place p2. Original nets allowed only one token to be added or removed from a place whenever a transition fired. Weighted Petri [305] nets are a generalization which allow multiple tokens to be added or removed from places.

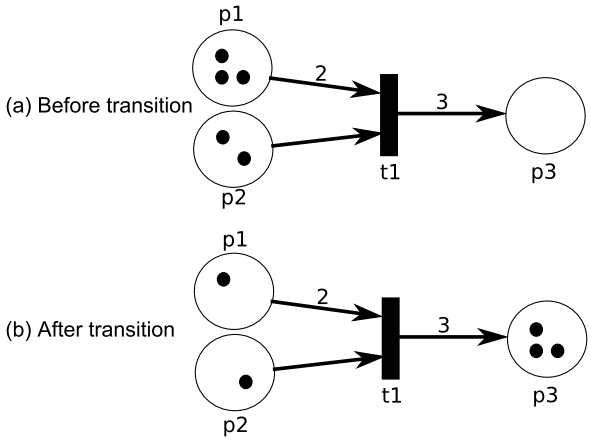

Fig. 11.2. Weighted Petri nets

Figure 11.2 shows a Weighted Petri net with the weights represented as positive integers labelling the edges. The edge from place p1 to transition t1 is given a weight of 2, the edge from place p2 to transition t1 is not labelled, implying a weight of 1, and the edge from t1 to p3 is given a weight of 3. When the transition t1 fires, two tokens are removed from p1, one token is removed from p2 and three tokens are added to p3.

There are other well-known extensions to the original Petri net model. These include colored Petri nets [306], timed Petri nets [307] and hierarchical Petri nets [308]. In traditional nets, the tokens have no types associated with them. The precondition for a transition to fire is that there be sufficient tokens available at the input places. The postconditions for a transition are that tokens be removed from the input places and added to the output places. With colored Petri nets, tokens are typed or *colored*. This is useful as the tokens in a Petri net model are usually modeling real-world objects that have associated attributes. The transitions in a colored Petri net can use the type and values of the consumed tokens to determine the type and values of the produced tokens.

When modeling real systems, it may be important to model temporal aspects of the system. In other words, there may be a need to model durations and delays. Timed Petri nets make this possible by associating time with tokens, places or transitions. For example, the model may be set up so that transitions take a certain amount of time to complete. When a transition is enabled, the tokens are removed

from the input places. After a certain time duration, tokens are added to the output places. A consequence is that the state of the system is not always clearly represented.

Petri nets for large systems can easily become very complex and difficult to analyze. This difficulty can be addressed using hierarchical Petri nets which allow a hierarchy of subnets to be constructed, each of which can be used to analyze one particular area of the system. Each subnet can be considered as a black box that may accept inputs from, and provide outputs to, other parts of the system being considered. It can be mathematically proven that the combination of subnets for a hierarchical Petri net have the same behavioral semantics as if the entire system were modeled as one very large single net. The main benefit they offer is the ease of use of Petri nets when modeling large and complex systems.

Modeling asynchronous distributed systems using Petri nets allows the model to be checked for a number of potentially undesirable properties. According to [309], these include:

- **Termination.** Does the Petri net terminate?
- **State reachability.** Are all possible states for the Petri net reachable?
- **Immediate reachability.** Is a particular state reachable when a specific transition fires?
- **Partial deadlock.** Is there a state where there is at least one transition that can never fire?
- **Deadlock.** Is there a state where no transition can fire?
- **Livelock.** Is there a set of states where the only transitions that can fire move between the states so that the Petri net never terminates?

11.2.4 Process Algebras

Process algebras (or process calculi) are algebraic languages that provide a formal foundation for modeling programs which can run concurrently in parallel, and which can interact with each other. In the case of such paralell systems, it's insufficient to say that each program can be simply modeled as an input/output function because the interaction between them affects their respective behaviours. Baeten [387] provides a pragmatic description by focussing on the individual definitions of the words "process" and "algebra". He points out that "process" refers to the behaviour of a system, or the total events or actions that the system can perform, the order in which they are executed and various aspects of this execution. In the context of modeling systems it is useful to keep the focus on certain essential aspects of the behaviour possibly ignoring other real-world considerations so that process models describe an observation of the behaviour of interest. The word "algebra" indicates using a generalized axiomatic approach in describing the process model. With a process modeled using algebraic equations, it becomes possible to apply algebraic laws to allow descriptions to be manipulated and analysed, and also provide a basis for formal reasoning about the process.

Petri nets preceded the conception of Process algebras by about a decade. The first Process algebra was devloped by Milner in the early 1970s and published as A Calculus for Concurrent Systems (CCS) [390]. Pi-calculus [391], which has become a popular Process algebra, has CCS as its theoretical starting point. In the examples later in this section, we use Pi-calculus as a representative process algebra. However, there are many others and a good starting point for further reading is in Baeten's work at [387].

Although Petri was the first person to develop models of interacting sequential processes, the focus of Process algebras is slightly different. A high-level difference is that Petri nets are bipartite graphs, while CCS (as a representative Process algebra) is a more textual, linear-like set of equations using an algebra that includes operators for concurrency, parallelism, conditions and functions (or data buffers). Van der Aalst [388] points out many notions for Petri nets have been translated into process algebra and vice versa. He argues that an important difference is that the notion of invariants devloped for Petri nets do not exist for Process algebra. In [389] the authors highlight that, although both approaches model concurrent systems, they tend to be used by different communities. Petri nets are popular with system and control engineers interested in issues around liveness and dynamic invariants of system design. Process algebras, such as CCS, are more popular with computer scientists who have some interest in liveness and invariance but are more interested in comparing the behaviours of systems. Another difference is that the Process algebras define systems as a collection of independent agents communicating with each other. Petri nets allow systems to be defined whose actions depend on internal and external inputs but it is not always easy to identify individual agents within the net.

As explained by Milner in his tutorial at [392], fundamental to the Process algebra of CCS, and more recently of Pi-calculus, is the notions of *naming*. In the basic version of Pi-calculus, there are only two types entities: *names* and *processes*. Names have no structure and there can be infinitely many of them. Processes, in Pi-calculus, are built from names using the syntax:

$$P ::= \Sigma_{i \in I}\, \pi_i \cdot P_i \;\mid\; P|Q \;\mid\; !P \;\mid\; (vx)P$$

There are four parts to the right hand side of this definition. Each part is separated by a large vertical bars representing the logical *OR* operator. These parts are described briefly below in Table 11.1 while a full description is available in [392].

Table 11.1. Parts of Calculus Definition

$\Sigma_{i \in I}\ \pi_i \cdot P_i$	The symbol pi is a prefix that represents an atomic action starting a process. I is an infinite prefixing set. There are two kinds of prefix. $x(y)$ which means input a name y along a link (or channel) x. This binds y in the prefixed process. $\bar{x}y$ which means output the name y along the link x. In this case, y is not bound to the process. x and $\bar{x}$ are both names referring to links. x is used for input while its co-name, $\bar{x}$, is used for output. The summation in the expression represents a process that is able to take part in exactly one of several alternatives for communication. The choice itself is not made by the process.
$P \mid Q$	The processes P and Q are concurrently active, can act independently and can communicate with eac other
$!P$	! is the replicator operator. This expression is shorthand for multiple copies of P running concurrently. (The calculus does not restrict this number.) Milner calls this "bang P"
$(\nu x)P$	Restricts the use of the name x exclusively to the process P. Milner calls this "new x in P".

The "**.**" (dot) operator indicates sequential actions. The final action for a process may be represented as a null action or **"O"** e.g. $x(y)$**.O**, however this is usually omitted in favour of $x(y)$. We now explain a simple example of a processs described by Milner using Pi-calculus.

$$\bar{x}y \mid x(u).\bar{u}v \mid \bar{x}z$$

This process is equivalent to three concurrent processes, $P \mid Q \mid R$, where P represents y available for output on channel x, Q represents u expected as input on channel x, and R represents z available for output on channel x. One of two communications can happen on channel x but not both. Consequently there are two possible outcomes for the result:

$$\mathbf{O} \mid \bar{y}v \mid \bar{x}z \quad \text{or} \quad \bar{x}y \mid \bar{z}v \mid \mathbf{O}$$

To see how these outcomes are derived, we look in more detail at the first of them. In P, y is output on the channel x (i.e $\bar{x}y$), and is accepted by Q as input along the channel x (i.e. $x(u)$). The subsequent action uses the name y as the channel to output the name v (i.e. $\bar{y}v$). If this first set of actions takes place then R ($\bar{x}z$) remains unchanged.

A relatively recent application of Process algebras is to Web Service composition. In Section 10.1.4, we described how there are two sides to service composition. The first is where components, represented as services, are put together one

after another, specified by control and data flow, to achive a particular task. In order to use such a composition, its not necessary to know its internals. The second aspect is the behaviour of the composition with respect to its requester. We pointed out that, in this way, service compositions can be seen to have both internal and external behaviour. Languages such as WSPBPEL provide a means of describing this behaviour structurally in tersm of XML. Process algebras (as do Petri nets) provide a formal language for describing the behaviours. One example of this application is in [393] where the authors focus on the use of Process algebra for simulation, property verification, and correctness of composition of Web Services. Another example in [394] describes a Process algebra called Finite State Process (FSP) for which the authors have developed a tool called LTSA-WS for the analysis of Web Service compositions described using WSBPEL.

11.3 Semantic Web Service Approaches

In this section, we look at the four leading ontology-based approaches for representing Semantic Web Services. These are OWL-S, SWSF, SAWSDL, WSDL-S and WSMO. In each case, the conceptual model is described and the languages used to express that model are explained.

11.3.1 OWL-S

Conceptual Model

OWL-S [310] is a Web Ontology Language (OWL) ontology, structured into three sub-ontologies, for describing different aspects of Semantic Web Services. The first aspect is the functionality a Web Service offers, including the constraints and non-functional properties that influence it. This is described using the ServiceProfile. Web Services enact their functionality through a behavioral model. Describing this is the aim of the ServiceModel. Finally, OWL-S seeks to build on top of WSDL and SOAP by mapping elements in the ServiceModel to elements in the WSDL description. This part of the OWL-S ontology is called the ServiceGrounding. We look at each of the three parts in the next paragraphs.

In OWL-S, the ServiceProfile describes what a Web Service does and provides the means by which the service can be advertised. As there is no distinction in the conceptual model of OWL-S between service requests and service provisions, the ServiceProfile is aimed equally at advertising services offered by providers and services sought by requesters. Owing to its genesis in the research area of artificial intelligence (AI), OWL-S defines the capability a service offers in terms of a state transition. It is possible to specify the inputs and outputs expected to be sent to and received from a service along with preconditions that must hold before the service can execute and the effects of the service executing. The intent is that along with arbitrary non-functional properties, this should be sufficient information for a dis-

covery agent to be able to decide if a desired ServiceProfile matches any of the ServiceProfiles in the set of candidate OWL-S Web Service descriptions available to it.

The ServiceModel is used to define the behavioral aspect of the Web Service. This part of the service is modeled as a process in the sense that a service requester can view the process description and understand how to interact with the service in order to access its functionality. In some ways, this process model can be considered as a partial workflow where the service requester provides the missing parts. The ServiceModel allows for the description of different types of services, atomic, abstract and composite. Atomic processes correspond to a single interaction with the service, e.g., a single operation in a WSDL document. Composite processes have multiple steps, each of which is an atomic process, connected by control and data flow. Simple processes are abstractions to allow multiple views on the same process. These can be used for the purposes of planning or reasoning. Simple processes are not invocable but are described as being *conceived* as representing single-step interactions. A simple process can be *realized by* an atomic process or *expanded to* a composite process.

The final part of the conceptual model is the ServiceGrounding, providing a link between the ServiceModel and the description of the concrete realization for a Web Service provided by WSDL. Atomic processes are mapped to WSDL operations, where the process inputs and outputs, described using OWL, are mapped to the operation inputs and outputs, described using XML Schema. It is possible that a single OWL-S Atomic Process can be mapped to many WSDL operations (although this is not usually the case). Composite processes, being composed of atomic processes, are grounded in the same way with the additional requirement of an OWL-S process engine to interpret the defined control and data flow.

In many ways OWL-S was the first consensus-based ontology for describing Semantic Web Services. It is the product of merging earlier research from two separate languages, DAML [291] and OIL [292], resulting in an ontology initially called DAML-S but later renamed to OWL-S to emphasize the perceived layering of the ontology on OWL (a W3C Recommendation). The actual use of the description logics variant, OWL-DL, as the ontology language for OWL-S has some unwanted side effects noted in detail in [311]. In particular, OWL-S does not comply with the OWL-DL specification, which places constraints on how OWL-S ontologies can be reasoned over. A second problem is that variables are not supported within OWL-DL but are necessary when combining data from multiple cooperating processes in OWL-S.

Language

Although primarily the OWL-S ontology is defined using the Web Ontology Language (OWL), OWL-S is actually a mixture of a number of languages. This breaks to some extent the claim for OWL-S that it is layered on top of OWL (and so a natural candidate for standardization). The reason for the language mixture is that

Web Services are inherently associated with distributed computing on the Web through process definition and execution. OWL is simply not designed for this purpose. Rather, it provides an upper ontology for defining conceptual models. In particular, to take advantage of the most commonly available implemented logical reasoners, OWL-DL is used to define the domain models used in the Semantic Web Service descriptions.

When describing logical expressions for the preconditions and results of ServiceProfiles or ServiceModels, the modeler has a choice. The Semantic Web Rules Language (SWRL) [312] and Resource Description Framework (RDF) [42] treat expressions as XML literals while the Knowledge Interchange Format (KIF) [313] or the Planning Domain Description Language (PDDL) [314] can be used for treating expressions as string literals.

11.3.2 SWSF

The establishment of the Semantic Web Services Framework (SWSF) [315] was motivated by the recognition of some shortcomings of OWL-S as a conceptual model for Semantic Web Services. At the time OWL-S was developed, attention was focussed on how an ontology for Web Services could be described using OWL. OWL itself is layered on top of the Resource Description Framework (RDF), and it was considered an elegant solution to add OWL-S as a further layer. A significant problem, as indicated in Section 11.3.1, is that OWL (or more precisely OWL-DL) is not well suited to describing processes. This situation is unsatisfactory as the functionality offered by Web Services can be considered as a *partial* process involving the operations that the Web Service makes available to a client application. The process description is partial as the client itself provides the complimentary activities when it interacts with the service.

SWSF was devised to provide a full conceptual model and language expressive enough to describe the process model of Web Services. There are two parts to the SWSF. The first is a conceptual model called the Semantic Web Services Ontology (SWSO) axiomatized using first order logic, and the second is a language called the Semantic Web Services Language (SWSL).

Conceptual Model

SWSO defines a conceptual model for Semantic Web Services with a deliberate focus on extending the work of OWL-S to interoperate with and provide semantics for industry process modeling formalisms like the Business Process Execution Language (BPEL). The first-order logic axiomatisation of SWSO is called FLOWS (First-Order Logic Ontology for Web Services) and is based on the Process Specification Language (PSL) [316], an international standard ontology for describing processes in domains of business, engineering and manufacturing. One of the intentions of PSL was to provide a common interlingua for the many existing process languages, allowing interoperability to be established between them. As the

number of conceptual models and languages for Semantic Web Services grows, there is a perceived need for such an umbrella formalism to facilitate interoperability in this area.

As mentioned, FLOWS is axiomatized in first-order logic and is expressed in a language called SWSL-FOL (Semantic Web Services Language for First-Order Logic). To enable logic-programming-based implementations and reasoning for SWSO, there is a second ontology available called ROWS (Rules Ontology for Web Services) and this is expressed in SWSL-Rules. ROWS is derived from FLOWS by a partial translation. The intent of the axiomatisation of ROWS is the same as that of FLOWS but in some cases it is weakened because of the lower expressiveness of the SWSL-Rules language compared to SWSL-FOL.

Service is the primary concept in SWSO with three top-level elements, derived from the three parts of the OWL-S ontology. These are Service Descriptors, Process Model and Grounding.

Service Descriptors. They provide a set of non-functional properties that a service may have. The FLOWS specification includes examples of simple properties such as the name, author, and textual description. The set is freely extensible to include the properties identified in other conceptual models such as WSMO non-functional properties or OWL-S service profile elements. Metadata specifications for online documents including Dublin Core[7] are also easily incorporated. Each property is modeled as a relation linking the property to the service. For example, Figure 11.3 shows FLOWS relations for service_name, version and reliability. Note that Web Service reliability is a subjective notion in the context of the quality-of-service (QoS) attributes a service may have. For it to be effective, a formal description of the meaning of reliability in Web Services is required. Some ongoing work in modeling this type of attribute using WSMO ontologies is described in [351].

```
name(service, service_name)
version(service, service_version)
reliability(service, service_reliability
```

Fig. 11.3. FLOWS Service Descriptor Properties

Process Model. The underlying objective of PSL is to provide a language and ontology that is expressive enough that all other process languages can be represented in it. If this is achieved then the integration of independent processes described with heterogeneous models becomes possible. FLOWS extends the PSL generic ontology for processes with two fundamental elements, especially to cater to Web Services:

- The structured notion of atomic processes as found in OWL-S

7. http://dublincore.org/ (Accessed September 10, 2007).

- Infrastructure for allowing various forms of data flow

The Process Model of FLOWS is organized as a layered extension of the PSL-OuterCore ontology. The primary layer is called FLOWS-Core and contains the two extensions just mentioned for Web Services. On top of this, five additional ontology modules are defined that are used to express different constraints on the occurrences of services and their subactivities. A simplified diagram of this layering is provided in Figure 11.4.

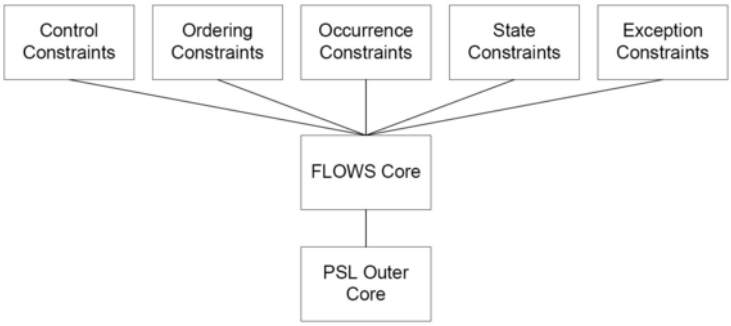

Fig. 11.4. FLOWS layered process model

As defined in the SWSF submission to the W3C submission, the layer has five additional ontologies:

- **Control Constraints** axiomatize the basic constructs common to workflow-style process models. In particular, the control constraints in FLOWS include the concepts from the process model of OWL-S.
- **Ordering Constraints** allow the specification of activities defined by sequencing properties of atomic processes.
- **Occurrence Constraints** support the specification of non-deterministic activities within services.
- **State Constraints** support the specification of activities triggered by states (of an overall system) that satisfy a given condition.
- **Exception Constraints** provide some basic infrastructure for modeling exceptions.

Four key terms defined by the FLOWS ontology are listed below:

- **Service.** A service is an object that can have an associated number of service descriptors as described above, and an activity that specifies the process model of the service.
- **Atomic Process.** An atomic process is generally a subactivity of the activity associated with a service. It is directly invocable, has no subprocesses and can be executed in a single step.
- **Message.** Messages have an associated message type and payload.

- **Channel.** A channel is an abstraction for an object that holds messages that have been sent but may not yet have been received. There is no restriction that all messages sent be associated with channels, but where this is the case there are additional axioms that must hold for the message.

Before leaving this brief description of the FLOWS Process Model, we draw attention to the fact that FLOWS allows the modeling of predicates or terms whose values may change in the course of an activity. The modeling elements are called *fluents* and can be imagined as providing a behavior similar to that of variables in a programming language, in that they allow processes to be chained together where a value from one process may be required by another. The absence of this was one of the observed drawbacks of the OWL-S process model.

Grounding. The SWSO approach to grounding follows very closely the grounding of OWL-S v1.1 to WSDL. The SWSO specification defines how the grounding must provide four things. These are:

- Mappings between the SWSO and WSDL messages patterns
- Mappings between message types as defined in SWSO and WSDL respectively
- Serialization from SWSO message types to the concrete message types defined by WSDL
- Deserialization from the concrete WSDL message types to the SWSO messages types

Language

We have already described that the Semantic Web Services Language (SWSL) comes in two variants: SWSL-FOL and SWSL-Rules. The starting point is SWSL-FOL which acts as a foundational ontology language with PSL as its foundation. SWS-Rules is derived as a partial translation to facilitate implementation and reasoning based on logic programming techniques.

Both variants share syntax but not the semantics of that syntax. In fact, neither language is a subset of the other, which means the two language variants are mutually incompatible (cannot be used together), which may somewhat complicate the understanding of how to use of SWSO/L. The modeler must decide which language best suits the purpose at hand. The decision is made simpler as each of the variants has a differing focus. SWSL-FOL is most useful for process-related descriptions while SWSL-Rules is geared toward the description of programming-like tasks such as discovery and contracting. Both variants comply with Web principles such as the use of URIs, integration with XML types and XML-compatible namespaces. Additionally both are layered languages where new features are incorporated at each layer.

A concise review of SWL-Rules is provided by the authors of [317]. As described in this report, SWSL-Rules is a logic programming language including features from Courteous logic programs [318], HiLog [319] and FLogic [320] and can be seen as both a specification and an implementation language. The SWSL-

Rules language provides support for service-related tasks such as discovery, contacting, and policy specification. It is a layered language as illustrated in Figure 11.5. The core of the SWSL Rules language is represented by a pure Horn subset. This subset is extended by adding features such as disjunction in the body and conjunction and implication in the head [321], or negation in the rule body interpreted as negation as failure (called NAF). Other extensions are (1) Courteous rules (Courteous), (2) HiLog, and (3) Frames.

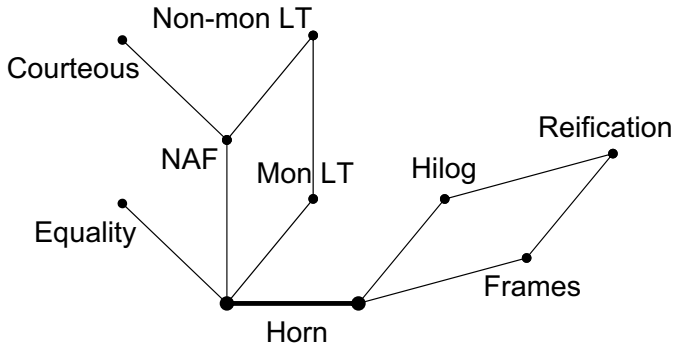

Fig. 11.5. SWSL-Rules Layers

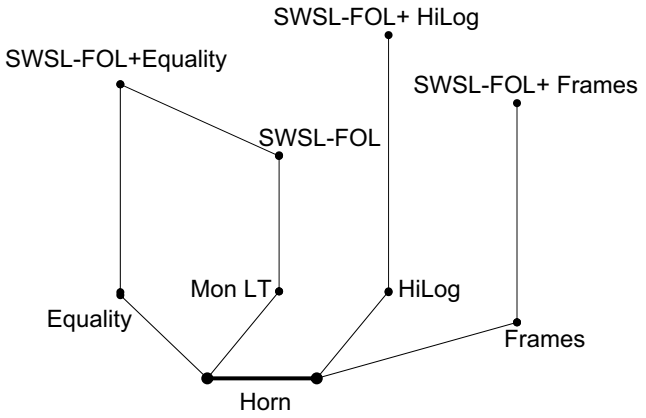

Fig. 11.6. Layers of SWSL-FOL and relationship to SWSL-Rules

On the other hand, SWSL-FOL, intended to describe the dynamic (process) aspect of services, is also layered. The bottom layer of Figure 11.6 shows the layers

of SWSL-Rules that have monotonic semantics and therefore can be extended to full first-order logic. The most basic extension is SWSL-FOL but Figure 11.6 also shows three other possible layered variants that can be achieved by the relevant extension. Theses are SWSL-FOL+Equality, SWSL-FOL+HiLog and SWSL-FOL+Frame.

11.3.3 WSDL-S

WSDL-S [322] is a lightweight approach for augmenting WSDL descriptions of Web Services with semantic annotations. It is a refinement of the work carried on by the METEOR-S group at the LSDIS Lab, Athens, Georgia,[8] to enable semantic descriptions of inputs, outputs, preconditions and effects of Web Service operations by taking advantage of the extension mechanism of WSDL. WSDL-S is agnostic to the ontology language and model used for the annotations of WSDL.

In the following paragraphs we take a look at the approach of WSDL-S, the conceptual model representing the approach and the extensions to the WSDL language that realize the semantic annotations.

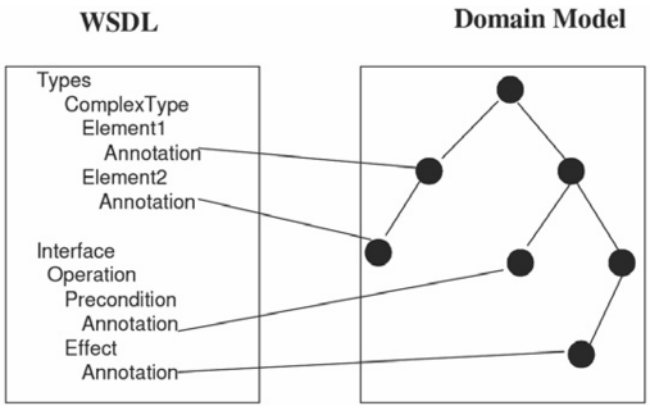

Fig. 11.7. Associating semantics with WSDL [322]

Approach. In contrast to the OWL-S, SWSO and WSMO, WSDL-S does not specify an ontology for the definition of Semantic Web Services. Rather, it takes a bottom-up approach with the appeal that potentially only a little additional effort on the part of service producers will provide a service description where the description of the data and operations of the service are bound to ontological concepts. WSDL-S intentionally builds directly on the existing Web Service technology stack.

8. http://lsdis.cs.uga.edu/ (Accessed September 10, 2007).

WSDL v1.1 allows for the definition of extension to its language. This is taken advantage of to provide an in-document link of certain WSDL elements to concepts in one or more ontologies (assuming that the concepts can be identified uniquely and that the links can be specified in legal XML). Figure 11.7 provides a high-level overview.

Embedding annotations into WSDL through legal language extensions does not affect the usage by the service provider of any other WS-* specifications or the usage of WSDL in the context of process description languages such as the Business Process Execution Language for Web Services (WSBPEL) [323]. Another feature is that where XML Schema is used as the data definition language for WSDL, it can be enhanced by linking XML Schema types to domain concepts either by a one-to-one mapping or through a transformation defined in a domain ontology.

Conceptual Model. WSDL-S defines its conceptual model using a simple XML Schema introducing five elements that extend WSDL. These are:

- **modelReference.** This is used for annotating both simpleTypes and complexTypes in XML Schema where there is a one-to-one mapping between the schema type and the ontological concept. For simpleTypes, it is a direct mapping. For complexTypes, it can be used in two ways, bottom-up and top-down. Bottom-level annotation involves describing every leaf element of the complexType with the modelReference attribute. Top-level annotation means that the complexType element itself is associated with a concept in the ontology. The assumption is that the subelements of the complexType will map directly to the sub-concepts and attributes of the domain concept.
- **schemaMapping.** Where there is no one-to-one mapping this attribute points to a transformation that links the XML Schema element to the ontology concept. For example, the value of the schemaMapping attribute might be a URI that identifies an XSLT transformation.
- **precondition.** At the level of a WSDL operation it is possible to point to a definition of the precondition that must hold before that operation can be executed. For simplicity only one precondition may be included and this may point to a set of logical expressions in the ontology language of choice.
- **effect.** Similar to preconditions, effects point to logical expressions that should hold after the execution of the service. In contrast to preconditions, WSDL-S allows for the definition of multiple-effect subelements of operations.
- **category.** This is adopted from OWL-S and is an extension to the WSDL Interface element of WSDL 2.0 (portType in WSDL 1.1). The intent is that category information can be included here that may be picked up by a Web Service registry implementation such as the one for UDDI.

Language. WSDL-S is defined using the XML-Schema listed in Figure 11.8

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
 targetNamespace="http://www.ibm.com/xmlns/stdwip/Web-services/WS-Semantics"
    xmlns:wssem="http://www.ibm.com/xmlns/stdwip/Web-services/WSSemantics"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

    <attribute name="modelReference" type="anyURI" use="optional"/>
    <attribute name="schemaMapping" type="anyURI" use="optional"/>

    <element name="category" maxOccurs="unbounded">
        <complexType>
            <complexContent>
              <extension base="wsdl:documented">
               <attribute name="categoryname" type="NCName" use="required"/>
              <attribute name="taxonomyURI" type="anyURI" use="required"/>
              <attribute name="taxonomyValue" type="String" use="optional"/>
              <attribute name="taxonomyCode" type="integer" use="optional"/>
               </extension>
            </complexContent>
        </complexType>
    </element>
    <element name = "precondition">
        <complexType>
            <complexContent
                <restriction base="anyType">
                    <xsd:attribute name="name" type="string" />
                    <attribute name="modelReference" type="anyURI" />
                    <attribute name="expression" type="string" />
                </restriction>
            </complexContent>
        </complexType>
    </element>
    <element name="effect">
        <complexType>
            <complexContent
                <restriction base="anyType">
                    <xsd:attribute name="name" type="string" />
                    <attribute name="modelReference" type="anyURI" />
                    <attribute name="expression" type="string" />
                </restriction>
           </complexContent>
        </complexType>
    </element>
</schema>
```

Fig. 11.8. XML Schema for WSDL-S

11.3.4 SAWSDL

The Semantic Annotations for WSDL (SAWSDL) working group, formed recently by the W3C, provides a W3C Candidate Recommendation for Semantic Web Services based on a simplified form of WSDL-S. This is in the form of an incremental bottom-up approach to Semantic Web Services where elements in WSDL documents are provided with semantic annotations through attributes provided using standard valid extensions to WSDL. The approach is agnostic to the ontological model used to define the semantics of annotated WSDL elements. From SAWSDL's perspective, the annotations are valued by URIs. SAWSDL, like WSDL-S is

targeted at WSDL v2.0 but it is also possible to use with WSDL v1.1 with an additional non-standard extension.

While WSDL-S specifies the attributes for modelReference, schemaMapping, precondition, effect and category, SAWSDL confines itself to attributes of modelReference and two specializations of schemaMapping, namely, liftingSchemaMapping and loweringSchemaMapping.

The modelReference attribute can be used to annotate XSD complex type definitions, simple type definitions, element declarations, and attribute declarations as well as WSDL interfaces, operations, and faults. The liftingSchemaMapping can be applied to XML Schema element declaration, complexType definitions and simpleType definitions. All attributes defined by SAWSDL are defined by the XML Schema, reproduced in Figure 11.9, to take a list of URIs as value.

```
<xs:schema
    targetNamespace="http://www.w3.org/2002/ws/sawsdl/spec/sawsdl#"
    xmlns="http://www.w3.org/2002/ws/sawsdl/spec/sawsdl#"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://www.w3.org/2006/01/wsdl">

    <xs:simpleType name="listOfAnyURI">
        <xs:list itemType="xs:anyURI"/>
    </xs:simpleType>

    <xs:attribute name="modelReference" type="listOfAnyURI" />
    <xs:attribute name="liftingSchemaMapping" type="listOfAnyURI" />
    <xs:attribute name="loweringSchemaMapping" type="listOfAnyURI" />

</xs:schema>
```

Fig. 11.9. SAWSDL XML Schema

11.3.5 WSMO, WSML and WSMX

Both WSMO and OWL-S address the same problem space. After identifying perceived fundamental drawbacks of the OWL-S approach, the WSMO working group was formed to devise a more complete conceptual model for describing Web Services. Conceptually WSMO, unlike OWL-S, explicitly models separate concepts for goals and Web Services. Additionally WSMO models mediators explicitly as first-class elements capable of bridging heterogeneity issues. In contrast, OWL-S does not explicitly model mediators. Rather, they are considered as specific types of services. A detailed discussion of this rationale is provided in [310].

Conceptual Model

Of the models for semantically annotating Web Services described so far, WSMO and OWL-S are the most closely related. Both aim at the provision of a comprehensive conceptual model for Semantic Web Services. The authors of WSMO describe how an important foundation point of the work on WSMO was the model provided by OWL-S but maintain that OWL-S has a number of serious fundamen-

tal flaws that give rise to problems when attempting to use the ontology in practice. These are described in detail in [324] but we will describe a subset of them in this description of the WSMO conceptual model and in the following section describing the languages for WSMO that provide the formal definition of semantics for the conceptual model. WSMO is predicated on a number of underlying principles as defined in [325]. These are:

- **Web compliance.** Every element of WSMO is identified using Unique Reference Identifiers (URIs). Namespaces are supported. WSMO can be serialized to XML, the language of the Web, and WSMO service descriptions ground to the Web Service Description Language (WSDL).
- **Ontology-based.** Ontologies are used to model every element of WSMO.
- **Strict decoupling.** WSMO resources are defined in isolation of each other. There is no assumption that every resource must be defined using the same ontologies.
- **Strong mediation.** Strict decoupling is made possible by the attention paid to mediation in the WSMO model. Mediators are top level modeling elements that are used to bridge interoperability issues between independent, heterogeneous WSMO resources.
- **Ontological separation of roles.** In WSMO the viewpoint of service requester and service provider are distinctly represented by the complementary concepts of goals and Web Services. This separation is adopted from the research in the problem-solving domain and is a clear point of distinction between the OWL-S and WSMO models.
- **Description versus implementation.** The ontological information model defined by WSMO and the additional associated functionality layers on top of existing Web Service implementation technology.
- **Service versus Web Service.** The definition of service within WSMO is a superset of Web Services described by the WSDL language. WSMO is designed to cover all types of service that may be available on the Web.

There are four top-level elements defined by WSMO as necessary for a comprehensive semantic description of Web Services. These are Ontologies, Web Services, goals and mediators. Each of these elements is represented as a class with a number of attributes. Attributes have their multiplicity set to multi-valued by default. If an attribute is single-valued, this is explicitly stated. All WSMO elements have the attribute *hasNonFunctionalProperty*. This allows for the assignment of any non-functional properties (e.g., related to quality-of-service or price, or metadata regarding the owner of the element) to any element. The following paragraphs provide a brief description of each of the top level WSMO elements.

Ontologies. They are used to define the information model for all aspects of WSMO. Compared to structural languages used to define taxonomies such as XML, ontologies allow for the formal definition of concepts and attributes in addition to restrictions and rules constraining them as well as functions and relations that range over them. Two key distinguishing features of ontologies are the princi-

ples of a *shared conceptualization* and a *formal semantics*. Ontologies are only useful if the meaning they express corresponds to a shared understanding by its users. Likewise, the strength of an ontology is that the semantics of its elements are machine-understandable, made possible through the provision of a mathematical base for the language used to express the ontology. Ontologies defined in WSMO are part of the MOF model layer.

Web Services. From a simplified perspective, WSMO Web Services are defined by the functional *capability* they offer and one or more *interfaces* that enable a client of the service to access that capability. Capability is one example of an attribute of a WSMO class (i.e., Web Service) that is single-valued. In WSMO a Web Service is defined as offering exactly one capability. The Web Service class also has attributes for mediators (used to bridge heterogeneity problems), non-functional properties (as described above) and ontologies that are imported (providing domain models for some part of the description). We will focus on the capability and interface descriptions as these constitute one area where the similarities and differences between WSMO and OWL-S are apparent.

The capability of a Web Service in the WSMO model defines the functionality that the service can provide when invoked by a service requester. It is defined using a state transition model (similarly to OWL-S but with more intuitive semantics). Prior to a Web Service invocation, *preconditions* define the required state of the information space available to the Web Service and *assumptions* define the state of the world outside that information space. An example of a precondition when using a Web Service to purchase goods is that a creditcard number be valid or that a postal code be valid for the delivery scope of the service. An example of an assumption is that the address provided actually exist. Preconditions and assumptions are defined using sentences in a logical language known as axioms. Depending on the language used, the axioms can be more or less expressive.

Correspondingly, when a service executes successfully, *postconditions* are used to define the state of the information space, and *effects* describe the state of the world outside the information space. For example, a postcondition might be that a shipment confirmation message be sent to the service requester and an effect might be that the goods be physically put in a container and shipped.

All four types of condition are optional in the capability description. The service can be considered as one or more state transitions that move from the state defined by the preconditions and assumptions to the state defined by the postconditions and effects. An application wishing to locate a service for a specific task uses the capability description of a WSMO service to determine if it offers the requisite functionality. Universally quantified shared variables are used to allow information to be shared between the four conditions allowed in capability descriptions.
For example, the listing below shows a Web Service capability from our running translational medicine example for a service providing Therapeutic Guidance. The capability states that on provision of patient information and a set of results corresponding to that patient, a collection of proposed therapies will be returned by the service. The scenario is fully described in context in Section 13.1.4.

```
capability _"http://TherapeuticGuidelines/capability"

precondition
   definedBy
      ?patient memberOf Patient and
      ?listTestResults memberOf ListResultsTests and
      ?listTestResults[patient hasValue ?patient].

postcondition
   definedBy
      ?listTherapies memberOf ListTherapies and
      ?listTherapies[patient hasValue ?patient].
...
```

While the capability defines what a service offers, the WSMO Web Service *interface* elements describe views of external parties on how they can interact with the service. These are is subdivided into two further elements, choreography and orchestration. The interface choreography element describes how a service requester can interact with the service to achieve its goal, including message exchange patterns, the process model supported and the definition of the information types exchanged at the interface. The interface orchestration element allows for the definition of a Web Service as an orchestration of other cooperating services (or goals, which we describe later). The idea is not that all (or indeed any) of the details of how a service achieves its capability be made public, but rather an explicitly described orchestration, including control and data flow and data definitions, allow the separation of the description of how the Web Service achieves its aims from its implementation. Both choreography and orchestration elements of WSMO Web Services are modeled using ontologized Abstract State Machines (ASMs) [327]. ASMs were chosen as a general model as they provide a minimal set of modeling primitives (no adhoc elements), are sufficiently expressive, and provide a rigid mathematical model for expressing dynamics.

The listing below shows a WSML interface description for a Web Service for getting guidance on tests to order for a patient in our translational medicine example. This WSML snippet is broken down and explained in detail in the context of the full example described in Section 13.1.6

```
wsmlVariant _"http://www.wsmo.org/wsml/wsml-syntax/wsml-rule"
...
interface GetTestOrderingGuidanceInterface
orchestration TestOrderingGuidanceOrchestration
  stateSignature GetTestOrderingGuidanceSignature

  /* Concepts used as input and output to the orchestration */
  in Patient withGrounding {
      _"http://.../RulesEngineService.wsdl#wsdl.interfaceMessageReference
            (GetTestOrderingGuidance/GetTests/In)"}
  in Patient withGrounding {
      _"http://.../DataIntegration.wsdl#wsdl.interfaceMessageReference
            (PatientHistory/GetCardiacHistory/In)"}
  shared PatientCardiacHistory withGrounding {
      _"http://.../RulesEngineService.wsdl#wsdl.interfaceMessageReference
            (GetTestOrderingGuidance/GetTests/In)"}
```

```
shared PatientCardiacHistory withGrounding {
    _"http://.../DataIntegration.wsdl#wsdl.interfaceMessageReference
          (PatientHistory/GetPatientCardiacHistory/Out)"}
out ListTests withGrounding {
    _"http://.../RulesEngineService.wsdl#wsdl.interfaceMessageReference
          (GetTestOrderingGuidance/GetTests/Out)"}
/* Concept used to define sequential control */
controlled ControlState

/* transition rules define the state changes of the orchestration */
transitionRules

if (
   ?patient memberOf Patient and
   ?patientCardiacHistory memberOf PatientCardiacHistory and
then
   add(_# memberOf ListTests)
   update(?cs[value hasValue RulesEngineServiceCalled])
endif

if (
   ?patient memberOf Patient and
   ?cs[value hasValue initialState] memberOf ControlState)
then
   add(_# memberOf PatientCardiacHistory)
   update(?cs[value hasValue PatientCardiacHistoryAvailable])
endif
```

Goals. WSMO goals are used to describe, from their own perspective, the aims service requesters have when they wish to interact with Web Services. The separation of goal and Web Service descriptions in WSMO is the realization of the objective to separate concerns. Service requesters are free to specify the services that they require in their own terms. This is one of the distinctions between OWL-S and WSMO. In OWL-S the service concept is used to describe both services and requests for services. Although from a modeling viewpoint WSMO goals and Web Services contain the same structure, they represent different perspectives in the conceptual model and for this reason are kept separate. Like Web Services, goals are defined with attributes for non-functional properties, imported ontologies, mediators, capabilities and interfaces. All of these attributes are defined from the perspective of what a service requester would like to get from a Web Service. The matching of goal and Web Service descriptions (usually referred to as service discovery) may require logical reasoning if syntactically different, but semantically similar, terms are used by the two parties. Semantic mismatches may be resolved using one or more of the mediator types defined by WSMO to cater to interoperability issues.

Mediators. The last of the four top-level elements of the WSMO conceptual model are mediators. They are used to bridge interoperability between any two WSMO elements. A number of distinctions are drawn in the WSMO mediator model. The first is between the description of a mediator and its implementation. While WSMO Web Service descriptions say nothing about how the services are implemented (they ground to WSDL for this), the same holds true for mediators (they can be optionally grounded in a goal, Web Service or another mediator). They describe the bridge that is required between any two elements. A second dis-

tinction is between the *kind of* mediation that is necessary for Semantic Web Services and the *types* of mediator that are defined by the WSMO model. The former breaks down to three varieties of mediation:

- **Data mediation.** Handle mismatches at the data definition level.
- **Protocol mediation.** Handle mismatches between message exchange protocols. This relates to the choreography descriptions of Web Services.
- **Process mediation.** Handle mismatches between heterogeneous business processes such as those defined by the RosettaNet[9] or ebXML[10] standards.

Other varieties of mediation may also become necessary over time. The list above is not considered exhaustive. The latter distinction is represented by the four types of mediator defined by WSMO:

- **OOMediators.** Cater to differences in the descriptions of data models defined by ontologies.
- **WGMediators.** Handle mismatches between the definition of a service request as expressed in a goal and the definition of an offered service as expressed in a Web Service
- **GGMediators.** While a repository of goals is already available, GGMediators allow goals to be linked together where there are differences in their descriptions. For example, say a goal is already known to match to a given Web Service; a match of a weaker goal to the same Web Service may be facilitated through a GGMediator.
- **WWMediator.** Analogous to the GGMediator. While a given Web Service already is known to match a specific goal, a weaker or stronger Web Service could also be matched to the same goal through the use of a bridging WWMediator.

Language

Earlier Section 5.2.4 included a subsection providing a description and detailed references for the WSML fanily of languages that provide formal semantics for the conceptual model of WSMO. The languages are layered to provide different levels of expressiveness for the semantics of WSMO depending on the reasoning requirements.

Execution Environment

The evaluation of the conceptual model and formal languages provided by WSMO and WSML respectively, is made easier by the availability of a reference imple-

9. http://www.rosettanet.org/ (Accessed May 20, 2008).

10. http://www.ebxml.org/ (Accessed May 20, 2008).

mentation. The Web Service Modeling Execution Environment (WSMX) [330] [331] provides middleware functionality designed to take advantage of the semantic annotations of Web Services using the WSMO model. The implemented WSMX architecture provides an approach to the automated discovery, composition, mediation and invocation of Semantic Web Services. Other tools exist based on the conceptual models described earlier in this chapter but none provide a single coordinated platform capable of tackling all aspects of Semantic Web Service execution. Figure 11.10 shows a high-level overview of the WSMX architecture.

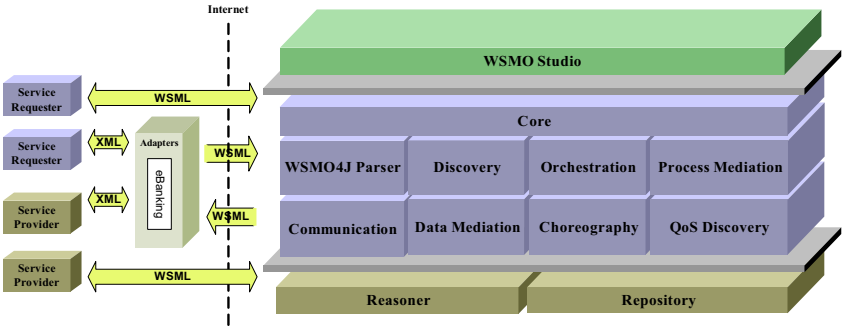

Fig. 11.10. WSMX Architecture

A detailed description of the WSMX architecture is available in [333]. Case-study-driven descriptions of its usage are available at [332]. In this section, we provide a brief description of the functionality of the various boxes in Figure 11.10 coupled with a description of some of the design decisions to create the platform to support this functionality.

The first point is that WSMX is intended as a middleware software layer *at the endpoints* of inter-service communications. This is an intent rather than a restriction. In other words, WSMX is not conceived as a third-party product that is independent of either a service requester or a provider but rather as a lightweight software layer that is positioned at the requester alone or at both the requester and the provider.

All information passed in and out of the WSMX boundary is represented in WSML. An adapter mechanism is provided to transform between non-WSML and WSML messages. All messages entering and leaving WSMX pass through the CommunicationManager which is responsible for handling any protocols relating to transport and communication. The WSMO4JParser is used to parse WSML descriptions to corresponding Java object models used as the internal data representation. Discovery takes care of matching goals to Web Services. The data and process mediation components take care of data, process and protocol heterogeneities where an appropriate mediator is available. The choreography and orchestration components are used to interpret and execute the abstract state machine models corresponding to the interface choreography and orchestration descrip-

tions. Quality of service discovery (QoSDiscovery) acts as a further match-making mechanism between goals and Web Services based on ontologically defined QoS attributes. On the bottom layer of the diagram, the WSML Reasoner acts at the heart of the platform, being necessary for logical reasoning of WSML descriptions for the discovery, mediation, choreography and orchestration functions. At the top of the diagram, WSMO Studio[11] and the Web Services Modeling Toolkit (WSMT)[12] are two alternatives for a WSMO modeling environment.

WSMX is implemented using an event-based messaging mechanism based on Java Management Extensions (JMX)[13] and JavaSpaces[14]. These provide a light-weight community standards mechanism that allows all of the WSMX components to be decoupled from each other. WSMX components are implemented in Java and interact with each other through an event-based publish-and-subscribe messaging system. More information on this and the open source implementation code of each component is available at the WSMX SourceForge project Web site.[15]

11.4 Reasoning with Web Service Semantics

Semantic annotation makes it possible for computers to understand the meaning of data and make more accurate decisions on how that data should be processed. When we talk about computers being able to understand data, we mean that the data is expressed in a language based on some type of formal logic that a computer can reason over. The computational device that carries out this task is usually referred to as a reasoner. In the last section, we have reviewed the state-of-the-art efforts for Semantic Web Service ontologies and identified the logical formalisms on which they are based. In this section, we discuss three particular areas where reasoning with Web Service semantics provides significant value. These are discovery, composition and mediation.

11.4.1 Discovery

Both Preist [334] and Baida et al. [335] distinguish between the concept of a service and a Web Service. They define a service as something of value in a particular domain of interest. Web Services are considered as the agents that provide the

11. http://www.wsmostudio.org/ (Accessed September 10, 2007).
12. http://sourceforge.net/projects/wsmt (Accessed September 10, 2007).
13. http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/ (Accessed September 10, 2007).
14. http://java.sun.com/developer/technicalArticles/tools/JavaSpaces/ (Accessed September 10, 2007).
15. http://sourceforge.net/projects/wsmx (Accessed September 10, 2007).

actual service, while the details of how to interact with the Web Service are described using WSDL and the messages exchanged with the Web Service are formed using SOAP. In a broad sense, Web Service discovery means finding a provider agent (Web Service) that can offer something of value (service) in a particular domain that is of interest to the requester. In [336], the authors point out that WSDL Web Service descriptions provide the technical details for invoking a set of possible *concrete* services. For example, the Amazon Web Service allows for the purchasing of books, DVDs and CDs (amongst other things). The WSDL does not include any details of available titles. A requester looking for the concrete service "sell me a book with the title "The Lord of the Rings" would not find a direct match based on the WSDL Web Service description. Rather, he (or an agent operating on his behalf) would abstract his request to a search for WSDL Web Services that sells books. Once located, a set of such Web Services may be interrogated to check if they offer that particular title (or offer some concrete service). This leads to two stages of discovery, pointed out in [337], each of which may be strengthened through the use of semantic annotations. The first involves abstracting specific client requests, e.g., *from buy The Lord of the Rings* to *buy a Book.* The second is refining the results of the first stage so that a match with the specific request can be made. This second stage will usually involve interaction with the Web Service via the described interface.

In the rest of this section we look at existing efforts for Semantic Web Service discovery, paying attention to the underlying requirements for reasoning. We first look at keyword-based discovery using UDDI and then, in turn, look at subsumption-based matching using Description Logics (DL), request rewriting with algorithms for *best profile covering*, process querying and object-based discovery.

Keyword-Based Discovery

Keyword-based discovery is the basis of the first wave of efforts involving Web Services and the UDDI registry specification. Initially, UDDI was used much like a white-pages listing of available Web Services. Loosely structured information regarding the provenance of the Web Service providers is provided through six specific UDDI concepts:

- **businessEntity**: information about the business
- **businessService**: more detail on the service being offered
- **bindingTemplates**: each one describes a technical entry point for the service
- **tModels**: information regarding particular standards or specifications used by the service
- **publisherAssertion**: declare relationships between business entities
- **operationInfo**: metadata regarding the information in the other five categories, e.g., the time and date they were created.

Keyword-based Web Service discovery usually associated with the use of UDDI relies on string-matching techniques, and very often, with visual human inspection of the information returned either through a graphical user interface on the UDDI registry or with the use of a UDDI API. In either case, logical reasoning is not used to match syntactically different but semantically similar terms. There have been some efforts to build on the UDDI specification through the use of semantic annotation of the information contained in registry entries, e.g., categories linked to ontological concepts. The ontologies used for annotations may, for example, be referenced in the tModels for the entries. This was discussed in [338].

As is the case for all efforts using semantic annotation to aid Web Service discovery, the type of reasoning that may be used is dependent on the choice of ontological language. In particular, we described in Section 11.3.2 on SWSF and in Section 11.3.5 on WSMO and WSML how the various types of underlying logical formalisms reflect the reasoning that may be applied.

Subsumption-Based Discovery

The conceptual model for the OWL-S profile includes concepts for input and outputs of a Web Service. The formal logical language used for profile descriptions is OWL-DL (Description Logics). This is designed for the representation of complex hierarchies of information. In the subsumption-reasoning approach of [367], an advertisement matches a request when all the outputs of the request are satisfied by the advertisement and all the inputs required by the advertisement are provided by the request. The reasoner can infer from the subsumption hierarchy of concepts if particular concepts match even where there are syntactic differences. The underlying concepts of the inputs and outputs are used by the reasoner when computing potential matches. The assumption is that all concepts used in the description of the profiles of both requests and advertisements are defined in a specified registry of OWL-DL ontologies. If concepts are included from unknown ontologies, the reasoner will not recognize them or be able to reason over them.

Similar to the query-rewriting approach to Semantic Web Service discovery described in the next section, subsumption-based reasoning allows for degrees of matching, i.e., matching that recognizes the degree of similarity between advertisements and requests. Examples of degrees of matching are: exact match, plug-in match, subsumption match. Additionally, other algorithms take into account of the distance between concepts in a taxonomy tree. The amount of flexibility built into this kind of discovery is at the discretion of the designers of the matching algorithm.

Request Rewriting (with Best Profile Covering)

This approach builds on subsumption-based reasoning over the inputs and outputs of OWL-S service advertisements and requests. It is described in detail in [339] The algorithm extracts the inputs and outputs of the request, looking for *a combi-*

nation of Web Services that satisfies as much as possible the required outputs of the query, and that requires as little as possible of any inputs not provided by the query. The previous approach looked only for matches between one service request and one advertisement. The request is essentially rewritten into a description of the conjunction of Web Services from known OWL-S ontologies. Best profile covering means a much greater degree of flexibility is allowed in the matching algorithm. Two concepts are defined, *Profile rest* (Pres) and *Profile miss* (Pmiss). Pres is defined as the difference between the outputs defined in the query service profile description and the outputs defined in the advertisement service profile description. Pmiss is defined as the inputs required by the rewritten query (in terms of available Web Services) and the inputs provided by the service request.

Roughly speaking, the difference between two descriptions A and B (written A — B) means all the information that is part of A but not a part of B. In Description Logics, A — B may be a set of descriptions that are not semantically equivalent. In [339], the assumption is made that semantic equivalence holds and further references to how this can be achieved are provided. Best profile cover is defined as the situation where the size of Pres and Pmiss are minimized.

The inputs and outputs of the service requests and advertisements are normalized into clauses (where each clause is of a known concept). The best profile covering problem is then reduced to an interpretation of hypergraphs by defining the difference between two semantic descriptions as a set difference operation between the sets of atomic clauses of two semantic descriptions. Hypergraph theory is used so that the problem of discovering which Web Services best cover the query may be resolved by finding the minimum transversal of a hypergraph with the minimum cost. A hypergraph is constructed where each vertex represents a Web Service and each edge represents a clause (A) of the normal description of the output of the query. The edge is populated by services that have a clause A' in their output that is semantically equivalent to A.

To determine semantic equivalence, reasoning is essentially based on subsumption and consistency checking but the matching algorithm additionally provides a global reasoning mechanism, a flexible matching that goes beyond subsumption tests, and effective computation of missed information.

State-based Discovery

In Section 11.3.5 on WSMO, we described how Web Service and goal capabilities are modeled using preconditions, postconditions, assumptions and effects. As with inputs, outputs, preconditions and effects of OWL-S (Section 11.3.1), this represents a model for describing the state of the world before and after the execution of a Web Service. State-based discovery, as described in [337] for WSMO, seeks to takes advantage of these descriptions to check if the states described in the service request and advertisement, before and after the Web Service execution, match each other.

A state determines the properties of the real world and the available information at some point in time. An abstract service is considered as a set of state transformations. As described earlier, a Web Service description may be considered as abstract as it usually does not describe a single concrete service (e.g., sell books vs. sell the book with title "The Lord of the Rings"). A concrete service can be modeled as a transformation from one particular state to another. In [337], the authors describe a formal model for WSMO Web Services and goals, and based on this present a conceptual model for service location with four stages:

- **Goal discovery:** Locate a predefined goal that fits the requester's desire. The predefined goal is an abstraction of the requester's desire in a more generic and reusable from.
- **Goal refinement:** The goal is refined taking account of the specific information provided in the service request.
- **Abstract service discovery:** Using the capability descriptions of the goal and available Web Service descriptions (capabilities contain the conditions that define the states for before and after execution), Web Services that may be able to fulfill the service request are located. At this point there is no guarantee that the abstract capability of matching services will be sufficient for the request.
- **Service contracting:** The located services will be checked for their ability to satisfy the request. This will usually involve invocation of the services.

Additionally, the paper describes how abstract services and Goals can be represented as sets of objects during the discovery phase. Objects are both the outputs and the effects that can be observed by a requester as a consequence of delivery of a service. This is the key part of the discovery algorithm where the other parts of the capability description are used during the service contracting phase.

The layered family of WSML languages can be employed when defining capabilities, such that a greater degree of logical inference is available to implementations of the service discovery algorithm. This was discussed earlier in Section 11.3.5.

Process-Based Querying

Another approach to Web Service discovery uses the process ontology segment of Semantic Web Service descriptions. This is an important aspect of OWL-S, WSMO and, in particular, SWSF. The process models are queried using a process query language to determine if specific service advertisements match service requests. Such an approach is described in detail in [340]. For this purpose, process models are decomposed into the following concepts, against each of which a query can be made:

- **Attributes**: textual characteristics of the process
- **Decomposition**: a process may be composed of other subprocesses

- **Resource flows**: all process steps have input and output ports through which resources, used by the process, can flow
- **Mechanisms**: resources that are used by the process as distinct from resources that are consumed or produced
- **Exceptions**: characteristics of process failures

11.4.2 Semantic Web Service Composition

There is a significant relationship between Semantic Web Service discovery and composition. In general, the algorithms for composition depend on the availability of a set of Web Services that, when composed, provide functionality that matches that required by the request. Further, in the course of composition, one or more of the matching techniques, described in the last section, will be necessary to determine if a specific service matches the requirements of a particular stage in a service composition. That said, there is a substantial body of research into composition including and predating Web Service technology.

In this section, we look at a sample of the state-of-the-art approaches to Semantic Web Service composition using inference engines to assist in the composition by reasoning over semantic annotations. Specifically, we look at composition planning, constrained object models [341], process-based composition and workflow approaches.

AI Planning

Planning is a research topic adopted from artificial intelligence (AI) concerned with the realization of strategies by intelligent agents where the solution to the strategy is determined at run time based on information represented using some formal language. This is valuable as changes to the set of available services and additional information can be taken into account by the inference engine at each step of the planning. Broadly speaking, an initial and a final state are provided along with information (and constraints on that information) of actions that are available to the agent. Two common, broad approaches are adopted, forward chaining and backward chaining. In forward chaining, the agent starts with the initial state, looking for an action that can move the solution closer to the final state based on the available information, e.g., what actions can be executed where the inputs to that action are available in the information space. The process *chains* forward until the final state can be reached. Backward chaining starts with the desired final state and works backward to the initial state. As the models for Semantic Web Services presented in this chapter pay special attention to the formalization of the data consumed and produced by Web Services, as well as the constraints on that data, planning techniques, based on logical inference engines, are seen as a strong proposal to the problem of Web Service composition. A comprehensive review of AI plan-

ning is beyond the scope of this chapter. To give an indication of the variety of approaches, we provide brief descriptions and references to additional material.

In the work of McIlraith and Son [342], the authors propose the modeling of service requests and advertisements in terms of first-order situation calculus. Requests are represented as generic procedures while services are represented as actions that either change the state of the world or the information space. The logic programming language Golog is adapted and extended as a natural formalism for representing and reasoning about service composition in this context.

An interesting link between the Semantic Web Service and AI communities is through the relationship of PDDL and DAML-S (the precursor of OWL-S). DAML-S was strongly influenced by PDDL, resulting in a straightforward mapping between the languages (with restrictions). Consequently, an approach to Web Service composition proposed in [314] is based on the translation of DAML-S descriptions to PDDL and reuse of the PDDL planners.

In [343], the authors describe how Hierarchical Task Planning (HTN) is especially suited to composition of Web Services described by OWL-S, as HTN places particular focus on task decomposition and precondition evaluation, concepts that tailor well to the OWL-S process descriptions. In [344], the meta-model for automated planning from AI and the meta-model for process-based service enactment are merged in an effort to overcome the predominantly static nature of process descriptions favored by industry, such as those defined using BPEL. Overcoming the challenges involved in merging the meta-models allows for more dynamic compositions that can be flexibly enacted. Enactment means that the composed process itself is determined at run time based on the semantic description of the input and output data and relevant constraints. Once the composed process has been established, services are located for each activity and it is verified that the overall process is executable.

Workflow and Business Processes Technology

A popular approach to the composition of Web Services, from an industrial point of view, is through the use of business process modeling (BPM) where each step of a process can be performed by the execution of a Web Service. BPM itself shares a lot of its underlying theory with workflow modeling. Van der Aalst [345] provides a critical comparison Web Service composition language using a set of workflow patterns as the evaluation criteria. For a process or workflow to be established, the stages have to be identified and suitable activities selected. A control flow needs to be defined to ensure the correct sequence of invocation of each activity. Data flow also needs to be defined so that the correct datatypes are used to transfer data from one activity to another. Van der Aalst notes that Web Service composition languages adopt most of the functionality of workflow systems but show increased expressiveness and in particular put additional focus on communication patterns. He also points out the desirability of providing formal semantics for composition languages through mappings to established process modeling formalisms.

The Web Services Business Process Execution Language (WSBPEL)[16] provides an XML-based language for defining business processes in terms of operations provided by Web Services with WSDL descriptions. Although popular and maturing, BPEL essentially is a static means of describing processes made up of Web Service compositions. However, there is significant research activity to merge the theoretical aspects of workflow (and by extension BPM) with the rich expressiveness of Semantic Web Service descriptions in languages like OWL-S and WSMO.

For example, in [346], a BPEL process is defined manually as a *skeleton*. All candidate services with semantic annotations that may be used by the process (there may be multiple candidates for each step) are verified and then registered in a service container. The skeleton process can then be configured to use different combinations of services for different scenarios. A programmatic interface is used to carry out the configuration. For example, in a process that involves booking flights online, one airline's service may be replaced by another's without the need to modify the skeleton business process. As the process models including the input and output messages of each service are semantically described, inference reasoning comes into play where there are differences in the required inputs and outputs of messages for the various services. The reasoning engine can check for semantic compatibility and adjust the configuration of the process accordingly. A similar approach is described for the eFlow platform in [347] where the composite service is modeled as a graph. The graph consists of nodes for services, events and decisions. Arcs joining the nodes denote execution dependencies. The service nodes can be configured to resolve to a concrete service implementation either at design time or run time.

A related approach to process-based Web Service composition is goal-based orchestration [348] using the WSMO conceptual model. The key idea is that each stage in a process can be represented by a WSMO goal rather than a specific service identifier. The goals are resolved to concrete services at run time by a suitable execution environment such as WSMX. A three-tier model is proposed that allows the design of processes through a visual tool that can be mapped to a formal workflow language. The workflow language has then a direct mapping to the Abstract State Machine (ASM) formalism used to describe service behavior in WSMO orchestrations.

11.4.3 Mediation

A frequent, unstated assumption when tackling Web Service discovery and composition is that all artifacts (service requests and advertisements) use a common conceptual model for defining data, processes and protocols. In real-world conditions, it is highly unlikely that business partners can agree on this level of uniformity in

16. http://www.oasis-open.org/committees/wsbpel (Accessed September 10, 2007).

advance. Even within a single organization, where there are multiple operational units, each unit may use independent, heterogeneous conceptual models for legacy applications. In such a situation, both discovery and composition of services is very difficult without a defined means to bridge interoperability issues. Mediation is the activity of mitigating the problems of interoperability through ontology alignment. It has its origins in the significant history of research in the database community into schema mapping.

The formal description of data and process as promoted by Semantic Web Service technology provides the basis for mediation. Subsumption-based reasoning is used in the case of languages based on description logics such as OWL-DL while logic programming is used by WSML-Flight and rule-based reasoning is used for languages such as SWSL-Rule and WSML-Rule (which extends WSML-Flight with function symbols). Of the Semantic Web Service conceptual models discussed in this chapter, only WSMO defines mediators as a top level element. The other Semantic Web Service ontologies also recognize the necessity of mediation but do not model it explicitly within their scope. The four categories of WSMO mediators were identified in Section 11.3.5. In particular, current WSMO research efforts focus on design for data mediation and process mediation.

As defined in [349], data mediation is based on the definition of a formal model for ontology mappings. Mappings are created and stored using a formal language. The mappings are applied as needed when an issue of heterogeneity occurs. For example, in the WSMX execution environment, a goal may be defined in terms of one ontology while a candidate Web Service may use another. During the matching phase, the data mediation component checks for mappings between the two ontologies and applies mappings only as necessary. This means that usually only a subset of the mappings that correspond to the concepts used is required, helping the efficiency of the operation. The assumption is that the mappings between the ontologies have already been created.

In WSMO, process mediation deals with solving mismatches between the choreographies of interacting partners [331]. In other words, it is required where the requester's choreography (goal choreography) and the service's choreography do not match. Mismatches can appear not only when the requester and the provider use different conceptualizations of a domain (in which case data mediation is required), but also if they have different requirements for the message exchange pattern they wish to follow [350]. Essentially this means that one of them expects to receive/send messages in a particular order while the other has different messages or a different message order. The role of the process mediator is to retain, postpone and rebuild messages that would allow the communication process to continue.

11.5 Clinical Use Case

Using Semantic Web technologies to share the formal definition of the meaning of data models across the Web, so that they can be flexibly and powerfully queried, only tackles part of the problem with integrating independent heterogeneous applications. Such applications (and systems) interact with each other on the basis of the behaviour that they expose at their interfaces. This is well-recognized across various domains of interest, including medicine, where there are several specifications defining datamodels and behaviour within each respective domain.

In the running example, threading through this book from the use-case introduced in Chapter 2 to the detailed description in Chapter 13, we fous on a translational medicine scenario involving two of these sets of specifications: CEN 13606 and HL7-CDA. We look at how modeling both the data and the behaviour, defined for each specification, semantically can enable services to be located and combined more flexibly.

Applying semantics to Web Services means being able to speicfy the meaning of both the data and the behaviour that Web Services expose at their interfaces. The novelty of Semantic Web Services is not that it is the first technology seeking to define such semantics but that it applies existing semantic modeling mechanisms to Web Services rather than requiring that a new technology stack be built from the ground up.

In the last eight years multiple languages have emerged for the description of different aspects of the Semantics required by Web Services. The fundamental aims of these languages are very similar but each one either targets different specific aspects of Web Services or seeks to correct perceived inconsistencies in earlier efforts. There is also a varying degree of available tool support.

We choose WSMO as the conceptual model for the detailed example in Chapter 13 because of its support for mediation, the clear separation of modeling and ontological constructs for service requesters and providers, and its rule-based approach for the definition of behavioural semantics. Additionally, there is an open source execution engine available called WSMX[17] available for WSMO against which the model can be tested. WSMO has its own native language called WSML which we use in the example as it uses a frame-based syntax that is reasonably reader-friendly. It's important to note that WSML also can be expressed in RDF and can use the RDF examples included in other parts of this book.

The combination of Goals and rule-based process definitions for the sample translational medicine workflow means that the process designers can focus on what they want the process to do without having to worry at that point about the design implementation. Each step in the process is modeled as a Goal to be resolved to the most suitable service at run-time. For example, one Goal models

17. WSMX source and binaries are available at http://sourceforge.net/projects/wsmx, (accessed May 26, 2008).

the need to get guidance on the ordering of tests based on the symtoms and medical hsitory of a patient. Another Goal models the need to get therapeutic guidance depending on results returned from clinical and laboratory tests.

Each step may involve one or more independent services with possibly independent data and behaviour models. Mediation based on formally defined mappings act as bridges. These mappings do not come for free and require a design effort from domain experts. However, as they are defined between industry standards at the conceptual (rather than at the data-instance) level, they provide an extensible, flexible basis for reuse across multiple scenarios.

11.6 Summary

In this chapter we have taken a look at the state-of-the-art approaches for providing semantic annotations of Web Service descriptions. We started by looking at the motivation for applying semantics to Web Services, discussing the drawbacks of XML as a description language, in terms of providing machine-understandable and unambiguous semantics. As Web Service descriptions focus on process models for interacting with the software applications made available on the Web, we examined various approaches to capturing behavioral semantics. These included Finite State Machines, Statecharts and Petri-Nets. In the section on Semantic Web Service approaches we described in detail the four current prominent efforts, OWL-S, WSMO, SWSF and WSDL-S. Finally, as a strong motivation for the use of Semantic Web Services is the possibility to use logical inference engines to reason over semantic descriptions, we looked at three particular aspects of Semantic Web Service usage that may require reasoning support. These are discovery, composition and mediation. In each case, we discussed the prevalent underlying theories, most of which predate the introduction of the Semantic Web Services terminology.